Precise Interprocedural Dataflow Analysis via Graph Reachability

Thomas Reps,[†] *Susan Horwitz,*[†] *and Mooly Sagiv*[†, ‡]
University of Wisconsin

Abstract

The paper shows how a large class of interprocedural dataflow-analysis problems can be solved precisely in polynomial time by transforming them into a special kind of graph-reachability problem. The only restrictions are that the set of dataflow facts must be a finite set, and that the dataflow functions must distribute over the confluence operator (either union or intersection). This class of problems includes—but is not limited to—the classical separable problems (also known as "gen/kill" or "bit-vector" problems)—*e.g.*, reaching definitions, available expressions, and live variables. In addition, the class of problems that our techniques handle includes many non-separable problems, including truly-live variables, copy constant propagation, and possibly-uninitialized variables.

Results are reported from a preliminary experimental study of C programs (for the problem of finding possibly-uninitialized variables).

1. Introduction

This paper shows how to find precise solutions to a large class of interprocedural dataflow-analysis problems in polynomial time. In contrast with *intra*procedural dataflow analysis, where "precise" means "meet-over-all-paths"[20], a precise *inter*procedural dataflow-analysis algorithm must provide the "meet-over-all-*valid*-paths" solution. (A path is *valid* if it respects the fact that when a procedure finishes it returns to the site of the most recent call [31,15,6,24,21,29]—see Section 2.)

Relevant previous work on *precise* interprocedural dataflow analysis can be categorized as follows:

- Polynomial-time algorithms for specific individual problems (*e.g.*, constant propagation [5,14], flow-sensitive summary information [6], and pointer analysis [24]).
- A polynomial-time algorithm for a limited class of problems: the *locally separable* problems (the interprocedural versions of the classical "bit-vector" or "gen-kill" problems), which include reaching definitions, available expressions, and live variables [22].
- Algorithms for a very general class of problems [10,31,21].

The work cited in the third category concentrated on generality and did not provide polynomial-time algorithms.

In contrast to this previous work, the present paper provides a polynomial-time algorithm for finding precise solutions to a general class of interprocedural dataflow-analysis problems. This class consists of all problems in which the set of dataflow facts D is a finite set and the dataflow functions (which are in $2^D \rightarrow 2^D$) distribute over the meet operator (either union or intersection, depending on the problem). We will call this class the *interprocedural, finite, distributive, subset problems*, or *IFDS problems*, for short. All of the locally separable problems are IFDS problems. In addition, many non-separable problems of practical importance are also IFDS problems—for example: truly-live variables [13], copy constant propagation [12, pp. 660], and possibly-uninitialized variables.

Our results are based on two insights:

(i) By restricting domains to be powersets of atomic dataflow facts and dataflow functions to be distributive, we are able to efficiently create simple representations of functions that summarize the effects of procedures (by supporting efficient lookup operations from input facts to output facts). For the locally separable problems, the representations of summary functions are sparse. This permits our algorithm to be as efficient as the most efficient previous algorithm for such problems, but without losing generality.

(ii) Instead of calculating the worst-case cost of our algorithm by determining the cost-per-iteration of the main loop and multiplying by the number of iterations, it is possible to break the cost of the algorithm down into three contributing aspects and bound the *total* cost of the operations performed for each aspect (see the Appendix).

The most important aspects of our work can be summarized as follows:

- In Section 3, we show that all IFDS problems can be solved precisely by transforming them into a special kind of graph-reachability problem: reachability along ***interprocedurally realizable paths***. In contrast with ordinary reachability problems in directed graphs (*e.g.*, transitive closure), realizable-path reachability problems involve some constraints on which paths are considered. A realizable path mimics the call-return structure of a program's execution, and only paths in which "returns" can be matched with corresponding "calls" are considered.
- In Section 4, we present a new polynomial-time algorithm for the realizable-path reachability problem. The algorithm runs in time $O(ED^3)$; this is ***asymptotically faster*** than the best previously known algorithm for the problem [16], which runs in time

---

[†]Work performed while visiting the Datalogisk Institut, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark.
[‡]On leave from IBM Scientific Center, Haifa, Israel.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants CCR-8958530 and CCR-9100424, by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937), by the Air Force Office of Scientific Research under grant AFOSR-91-0308, and by a grant from Xerox Corporate Research.

Authors' address: Computer Sciences Department; Univ. of Wisconsin; 1210 West Dayton Street; Madison, WI 53706; USA.
Electronic mail: {reps, horwitz, sagiv}@cs.wisc.edu.

$$O(D^3 \sum_p Call_p E_p + D^4 \sum_p Call_p^3).$$

- As discussed in Section 5, the new realizable-path reachability algorithm is ***adaptive***, with asymptotically better performance when applied to common kinds of problem instances that have restricted form. For example, there is an asymptotic improvement in the algorithm's performance for the common case of locally separable problems. Our work generalizes that of Knoop and Steffen [22] in the sense that our algorithm handles a much larger class of problems, yet on the locally separable problems the algorithm runs in the same time as that used by the Knoop-Steffen algorithm—$O(ED)$.
- Imprecise (overly conservative) answers to interprocedural dataflow-analysis problems could be obtained by treating each interprocedural dataflow-analysis problem as if it were essentially one large intraprocedural problem. In graph-reachability terminology, this amounts to considering all paths rather than considering only the interprocedurally realizable paths. For the IFDS problems, ***we can bound the extra cost needed to obtain the more precise (realizable-path) answers***. In the important special case of locally separable problems, there is no "penalty" at all—both kinds of solutions can be obtained in time $O(ED)$. In the distributive case, the penalty is a factor of D: the running time of our realizable-path reachability algorithm is $O(ED^3)$, whereas all-paths reachability solutions can be found in time $O(ED^2)$. However, in the preliminary experiments reported in Section 6, which involve examples where D is in the range 70-142, the penalty observed is at most a factor of 3.4.
- Callahan has given algorithms for several "interprocedural flow-sensitive side-effect problems" [6]. Although these problems are (from a certain technical standpoint) of a slightly different character from the IFDS dataflow-analysis problems, with small adaptations the algorithm from Section 4 can be applied to these problems and is ***asymptotically faster*** than the algorithm given by Callahan. In addition, our algorithm handles a natural generalization of Callahan's problems (which are locally separable problems) to a class of distributive flow-sensitive side-effect problems. (This and other related work is described in Section 7.)
- The realizable-path reachability problem is also the heart of the problem of interprocedural program slicing, and the fastest previously known algorithm for the problem is the one given by Horwitz, Reps, and Binkley [16]. The realizable-path reachability algorithm described in this paper yields an ***improved interprocedural-slicing algorithm***—one whose running time is asymptotically faster than the Horwitz-Reps-Binkley algorithm. This algorithm has been found to run six times as fast as the Horwitz-Reps-Binkley algorithm [28].
- Our dataflow-analysis algorithm has been implemented and used to analyze several C programs. Preliminary experimental results are reported in Section 6.

Space constraints have forced us to treat some of the above material in an abbreviated form. Full details—including proofs of all theorems stated in the paper—as well as a great deal of additional material, can be found in [27].

2. The IFDS Framework for Distributive Interprocedural Dataflow-Analysis Problems

The IFDS framework is a variant of Sharir and Pnueli's "functional approach" to interprocedural dataflow analysis [31], with an extension similar to the one given by Knoop and Steffen in order to handle programs in which recursive procedures have local variables and parameters [21]. These frameworks generalize Kildall's concept of the "meet-over-all-paths" solution of an *intra*procedural dataflow-analysis problem [20] to the "meet-over-all-valid-paths" solution of an *inter*procedural dataflow-analysis problem.

The IFDS framework is designed to be as general as possible (in particular, to support languages with procedure calls, parameters, and both global and local variables). Any problem that can be specified in this framework can be solved efficiently using our algorithms; semantic correctness is an orthogonal issue. A problem designer who wishes to take advantage of our results has two obligations: (i) to encode the problem so that it meets the conditions of our framework; (ii) to show that the encoding is consistent with the programming language's semantics [9,10]. Encoding a problem in the IFDS framework may involve some loss of precision. For example, in languages in which parameters are passed by reference there may be a loss of precision for problem instances in which there is aliasing. However, the process of finding the solution to the resulting IFDS problem introduces no further loss of precision.

To specify the IFDS framework, we need the following definitions:

Definition 2.1. In the IFDS framework, a program is represented using a directed graph $G^* = (N^*, E^*)$ called a ***supergraph***. G^* consists of a collection of flow graphs $G_1, G_2, \cdots$ (one for each procedure), one of which, G_{main}, represents the program's main procedure. Each flowgraph G_p has a unique ***start*** node s_p, and a unique ***exit*** node e_p. The other nodes of the flowgraph represent the statements and predicates of the procedure in the usual way, except that a procedure call is represented by two nodes, a ***call*** node and a ***return-site*** node. (The sets of call and return-site nodes of procedure p will be denoted by $Call_p$ and Ret_p, respectively; the sets of all call and return-site nodes in the supergraph will be denoted by $Call$ and Ret, respectively.)

In addition to the ordinary intraprocedural edges that connect the nodes of the individual flowgraphs, for each procedure call, represented by call-node c and return-site node r, G^* has three edges:

- An intraprocedural ***call-to-return-site*** edge from c to r;
- An interprocedural ***call-to-start*** edge from c to the start node of the called procedure;
- An interprocedural ***exit-to-return-site*** edge from the exit node of the called procedure to r. □

(The call-to-return-site edges are included so that the IFDS framework can handle programs with local variables and parameters. The dataflow functions on call-to-return-site and exit-to-return-site edges permit the information about local variables that holds at the call site to be combined with the information about global variables that holds at the end of the called procedure.)

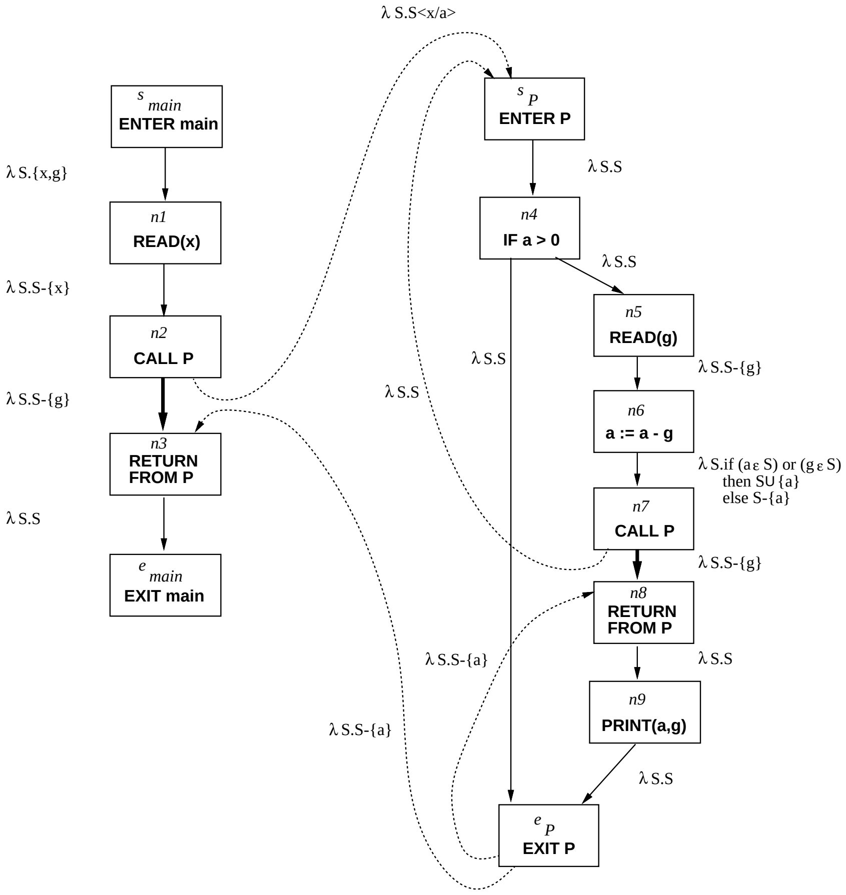
When discussing time and space requirements, we use the name of a set to denote the set's size. For example, we

```
declare g: integer

program main
begin
  declare x: integer
  read(x)
  call P(x)
end


procedure P(value a : integer)
begin
  if (a > 0) then
    read(g)
    a := a − g
    call P(a)
    print(a, g)
  fi
end
```

(a) Example program



(b) Its supergraph G^*

Figure 1. An example program and its supergraph G^*. The supergraph is annotated with the dataflow functions for the "possibly-uninitialized variables" problem. The notation S<x/a> denotes the set S with x renamed to a.

use *Call*, rather than $|Call|$, to denote the number of call nodes in graph G^*. (We make two small deviations from this convention, using N and E to stand for $|N^*|$ and $|E^*|$, respectively.)

Example. Figure 1 shows an example program and its supergraph G^*. □

Definition 2.2. A ***path*** of length j from node m to node n is a (possibly empty) sequence of j edges, which will be denoted by $[e_1, e_2, \cdots, e_j]$, such that for all i, $1 \le i \le j-1$, the target of edge e_i is the source of edge e_{i+1}. □

The notion of an "(interprocedurally) valid path" captures the idea that not all paths in G^* represent potential execution paths:

Definition 2.3. Let each call node in G^* be given a unique index i. For each such indexed call node c_i, label c_i's outgoing call-to-start edge by the symbol "$(_i$". Label the incoming exit-to-return-site edge of the corresponding return-site node by the symbol "$)_i$".

For each pair of nodes m, n in the same procedure, a path from m to n is a ***same-level valid path*** iff the sequence of labeled edges in the path is a string in the language of balanced parentheses generated from nonterminal *matched* by the following context-free grammar:

$$
\begin{array}{llr}
matched \;\rightarrow\; (_i \;\; matched \;\;)_i \;\; matched & & \text{for } 1 \le i \le Call \\
\qquad\qquad\;\; \mid \;\; \varepsilon & &
\end{array}
$$

For each pair of nodes m, n in supergraph G^*, a path from m to n is a ***valid path*** iff the sequence of labeled edges in the path is a string in the language generated from nonterminal *valid* in the following grammar (where *matched* is as defined above):

$$
\begin{array}{llr}
valid \;\rightarrow\; valid \;\; (_i \;\; matched & & \text{for } 1 \le i \le Call \\
\qquad\quad \mid \;\; matched & &
\end{array}
$$

We denote the set of all valid paths from m to n by IVP(m, n). □

In the formulation of the IFDS dataflow-analysis framework (see Definitions 2.4–2.6 below), the same-level valid paths from m to n will be used to capture the transmission of effects from m to n, where m and n are in the same procedure, via sequences of execution steps during which the call stack may temporarily grow deeper—because of calls—but never shallower than its original depth, before eventually returning to its original depth. The valid paths from s_{main} to n will be used to capture the transmission of effects from s_{main}, the program's start node, to n via some sequence of execution steps. Note that, in general, such an execution sequence will end with some number of activation records on the call stack; these correspond to "unmatched" $(_i$'s in a string of language $L(valid)$.

Example. In supergraph G^* shown in Figure 1, the path $[s_{main} \rightarrow n1,\ n1 \rightarrow n2,\ n2 \rightarrow s_p,\ s_p \rightarrow n4,\ n4 \rightarrow e_p,\ e_p \rightarrow n3]$ is a (same-level) valid path; however, the path $[s_{main} \rightarrow n1,\ n1 \rightarrow n2,\ n2 \rightarrow s_p,\ s_p \rightarrow n4,\ n4 \rightarrow e_p,\ e_p \rightarrow n8]$ is not a valid path because the return edge $e_p \rightarrow n8$ does not correspond to the preceding call edge $n2 \rightarrow s_p$. □

We now define the notion of an instance of an IFDS problem:

Definition 2.4. An ***instance*** *IP* of an ***interprocedural, finite, distributive, subset problem*** (or ***IFDS problem***, for short) is a five-tuple, $IP = (G^*, D, F, M, \sqcap)$, where

(i) G^* is a supergraph as defined in Definition 2.1.

(ii) D is a finite set.

(iii) $F \subseteq 2^D \rightarrow 2^D$ is a set of distributive functions.

(iv) $M : E^* \rightarrow F$ is a map from G^*'s edges to dataflow functions.

(v) The meet operator $\sqcap$ is either union or intersection.□

In the remainder of the paper we consider only IFDS problems in which the meet operator is union. It is not hard to show that IFDS problems in which the meet operator is intersection can always be handled by dualizing (*i.e.*, by transforming such a problem to the complementary union problem). Informally, if the "must-be-X" problem is an intersection IFDS problem, then the "may-not-be-X" problem is a union IFDS problem. Furthermore, for each node $n \in N^*$, the solution to the "must-be-X" problem is the complement (with respect to D) of the solution to the "may-not-be-X" problem.

Example. In Figure 1, the supergraph is annotated with the dataflow functions for the "possibly-uninitialized variables" problem. The "possibly-uninitialized variables" problem is to determine, for each node $n \in N^*$, the set of program variables that may be uninitialized just before execution reaches n. A variable x is possibly uninitialized at n either if there is an x-definition-free valid path to n or if there is a valid path to n on which the last definition of x uses some variable y that itself is possibly uninitialized. For example, the dataflow function associated with edge $n6 \rightarrow n7$ shown in Figure 1 adds a to the set of possibly-uninitialized variables if either a or g is in the set of possibly-uninitialized variables before node $n6$. □

To simplify the presentation, we assume in Definition 2.4 that there is a single global space of dataflow facts, D. This assumption is made strictly for expository purposes; the more general setting, in which for each procedure p there is a (possibly) different space of dataflow facts, D_p, presents no additional difficulties [27]. Our implementation of the IFDS framework, discussed in Section 6, supports the more general setting.

Definition 2.5. Let $IP = (G^*, D, F, M, \sqcap)$ be an IFDS problem instance, and let $q = [e_1, e_2, \cdots, e_j]$ be a non-empty path in G^*. The ***path function*** that corresponds to q, denoted by pf_q, is the function $pf_q =_{df} f_j \circ \cdots \circ f_2 \circ f_1$, where for all i, $1 \le i \le j$, $f_i = M(e_i)$. The path function for an empty path is the identity function, $\lambda x.x$. □

Definition 2.6. Let $IP = (G^*, D, F, M, \sqcap)$ be an IFDS problem instance. The ***meet-over-all-valid-paths*** solution to IP consists of the collection of values MVP_n defined as follows:

$$MVP_n = \bigsqcap_{q \in \mathrm{IVP}(s_{main},\, n)} pf_q(\top) \qquad \text{for each } n \in N^*.$$ □

3. Interprocedural Dataflow Analysis as a Graph-Reachability Problem

3.1. Representing Distributive Functions

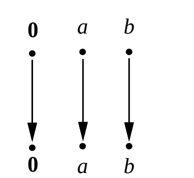
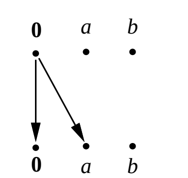
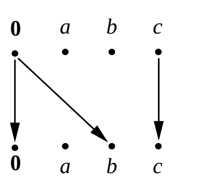
In this section, we show how to represent distributive functions in $2^D \rightarrow 2^D$ in a compact fashion—each function can be represented as a graph with at most $(D+1)^2$ edges (or, equivalently, as an adjacency matrix with $(D+1)^2$ entries). Throughout this section, we assume that f and g denote functions in $2^D \rightarrow 2^D$ and that f and g distribute over $\cup$.

Definition 3.1. The ***representation relation of f***, $R_f \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$, is a binary relation (*i.e.*, graph) defined as follows:

$$\begin{aligned} R_f =_{df}\ & \{(\mathbf{0}, \mathbf{0})\} \\ & \cup\ \{(\mathbf{0}, y) \mid y \in f(\emptyset)\} \\ & \cup\ \{(x, y) \mid y \in f(\{x\}) \text{ and } y \notin f(\emptyset)\}. \end{aligned}$$ □

R_f can be thought of as a graph with $2(D+1)$ nodes, where each node represents an element of D (except for the two $\mathbf{0}$ nodes, which (roughly) stand for $\emptyset$).

Example. The following table shows three functions and their representation relations:

id: $2^{\{a,b\}} \rightarrow 2^{\{a,b\}}$ $\mathbf{id} = \lambda S.S$	**a**: $2^{\{a,b\}} \rightarrow 2^{\{a,b\}}$ $\mathbf{a} = \lambda S.\{a\}$	**f**: $2^{\{a,b,c\}} \rightarrow 2^{\{a,b,c\}}$ $\mathbf{f} = \lambda S.(S - \{a\}) \cup \{b\}$
		

Note that one consequence of Definition 3.1 is that there is never an edge of the form $(x, \mathbf{0})$, where $x \in D$.

Another consequence of Definition 3.1 is that edges in representation relations obey a kind of "subsumption property". That is, if there is an edge $(\mathbf{0}, y)$, for $y \in (D \cup \{\mathbf{0}\})$, there is never an edge (x, y), for any $x \in D$. For example, in constant-function **a**, edge $(\mathbf{0}, a)$ subsumes the need for edges (a, a) and (b, a). □

Representation relations—and, in fact, all relations in $(D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$—can be interpreted as functions in $2^D \rightarrow 2^D$, as follows:

Definition 3.2. Given a relation $R \subseteq (D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$, its ***interpretation*** $[\![R]\!]: 2^D \rightarrow 2^D$ is the function defined as follows:

$$\begin{aligned} [\![R]\!] =_{df}\ & \lambda X.(\{y \mid \exists x \in X \text{ such that } (x, y) \in R\} \\ & \cup \{y \mid (\mathbf{0}, y) \in R\}) - \{\mathbf{0}\}. \end{aligned}$$ □

Theorem 3.3. $[\![R_f]\!] = f$.

Our next task is to show how the relational composition of two representation relations R_f and R_g relates to the function composition $g \circ f$.

Definition 3.4. Given two relations $R_f \subseteq S \times S$ and $R_g \subseteq S \times S$, their ***composition*** $R_f; R_g \subseteq S \times S$ is defined as follows:

$$R_f; R_g =_{df} \{ (x, y) \in S \times S \mid \exists z \in S \text{ such that } (x, z) \in R_f \text{ and } (z, y) \in R_g \}. \qquad \square$$

Theorem 3.5. *For all $f, g \in 2^D \to 2^D$, $[\![R_f; R_g]\!] = g \circ f$.*

Definition 3.4 and Theorem 3.5 imply that the composition of any two distributive functions in $2^D \to 2^D$ can also be represented by a graph (relation) with at most $(D+1)^2$ edges. In other words, the distributive functions in $2^D \to 2^D$ are "compressible": there is a bound on the size of the graph needed to represent any such function *as well as the **composition** of any two such functions!*

Corollary 3.6. *Given a collection of functions $f_i \colon 2^D \to 2^D$, for $1 \le i \le j$,*

$$f_j \circ f_{j-1} \circ \cdots \circ f_2 \circ f_1 = [\![R_{f_1}; R_{f_2}; \cdots ; R_{f_j}]\!].$$

3.2. From Dataflow-Analysis Problems to Realizable-Path Reachability Problems

In this section, we show how to convert IFDS problems to "realizable-path" graph-reachability problems. In particular, for each instance *IP* of an IFDS problem, we construct a graph $G^{\#}_{IP}$ and an instance of a realizable-path reachability problem in $G^{\#}_{IP}$. The edges of $G^{\#}_{IP}$ correspond to the representation relations of the dataflow functions on the edges of G^*. Because of the relationship between function composition and paths in composed representation-relation graphs (Corollary 3.6), the path problem can be shown to be equivalent to *IP*: dataflow-fact *d* holds at supergraph node *n* iff there is a "realizable path" from a distinguished node in $G^{\#}_{IP}$ (which represents the fact that $\emptyset$ holds at the start of procedure *main*) to the node in $G^{\#}_{IP}$ that represents

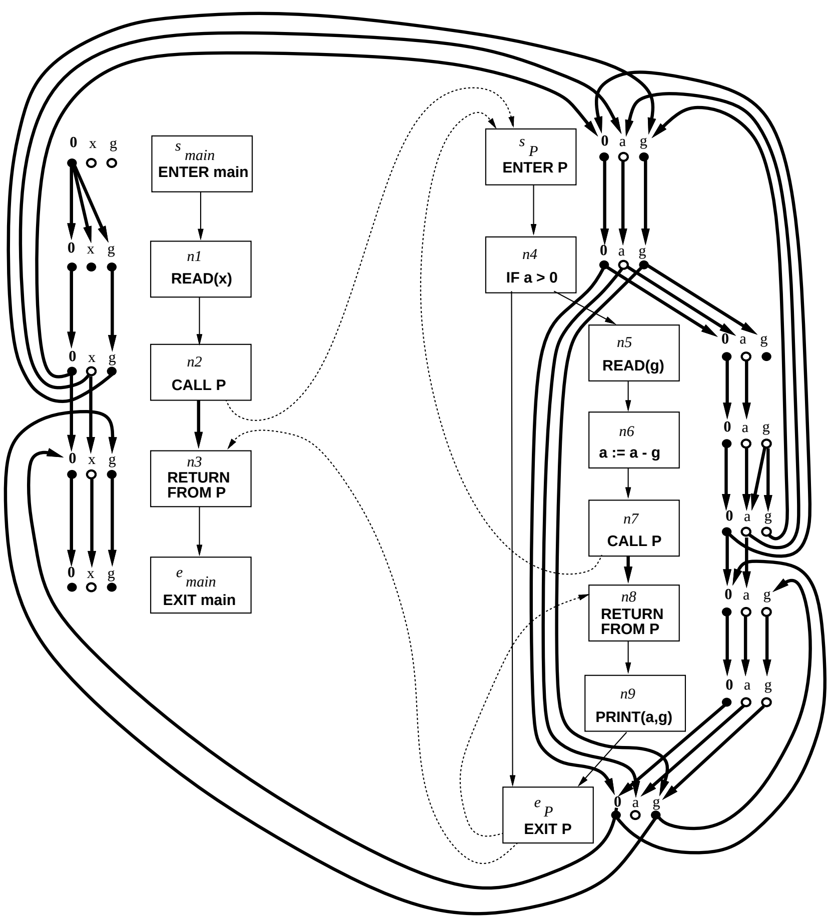


Figure 2. The exploded supergraph that corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 1. Closed circles represent nodes of $G^{\#}_{IP}$ that are reachable along realizable paths from $\langle s_{main}, \mathbf{0} \rangle$. Open circles represent nodes not reachable along such paths.

fact d at node n (see Theorem 3.8).

Definition 3.7. Let $IP = (G^*, D, F, M, \cup)$ be an IFDS problem instance. We define the ***exploded supergraph*** for IP, denoted by $G^\#_{IP}$, as follows:

$$G^\#_{IP} = (N^\#, E^\#), \text{ where}$$
$$N^\# = N^* \times (D \cup \{ \mathbf{0} \}),$$
$$E^\# = \{ \langle m, d_1 \rangle \to \langle n, d_2 \rangle \mid (m, n) \in E^* \text{ and } (d_1, d_2) \in R_{M(m,n)} \}. \quad \square$$

The nodes of $G^\#_{IP}$ are pairs of the form $\langle n, d \rangle$; each node n of N_p is "exploded" into $D + 1$ nodes of $G^\#_{IP}$. Each edge e of E^* with dataflow function f is "exploded" into a number of edges of $G^\#_{IP}$ according to representation relation R_f. Dataflow-problem IP corresponds to a single-source "realizable-path" reachability problem in $G^\#_{IP}$, where the source node is $\langle s_{main}, \mathbf{0} \rangle$.

Example. The exploded supergraph that corresponds to the instance of the "possibly-uninitialized variables" problem shown in Figure 1 is shown in Figure 2. $\square$

Throughout the remainder of the paper, we use the terms "(same-level) realizable path" and "(same-level) valid path" to refer to two related concepts in the exploded supergraph and the supergraph. For both "realizable paths" and "valid paths", the idea is that not every path corresponds to a potential execution path: the constraints imposed on paths mimic the call-return structure of a program's execution, and only paths in which "returns" can be matched with corresponding "calls" are permitted. However, the term "realizable paths" will always be used in connection with paths in the exploded supergraph; the term "valid paths" will always be used in connection with paths in the supergraph.

We now state the main theorem of this section, Theorem 3.8, which shows that an IFDS problem instance IP is equivalent to a realizable-path reachability problem in graph $G^\#_{IP}$:

Theorem 3.8. *Let $G^\#_{IP} = (N^\#, E^\#)$ be the exploded supergraph for IFDS problem instance $IP = (G^*, D, F, M, \cup)$, and let n be a program point in N^*. Then $d \in MVP_n$ iff there is a realizable path in graph $G^\#_{IP}$ from node $\langle s_{main}, \mathbf{0} \rangle$ to node $\langle n, d \rangle$.*

The practical consequence of this theorem is that we can find the meet-over-all-valid-paths solution to IP by solving a realizable-path reachability problem in graph $G^\#_{IP}$.

Example. In the exploded supergraph shown in Figure 2, which corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 1, closed circles represent nodes that are reachable along realizable paths from $\langle s_{main}, \mathbf{0} \rangle$. Open circles represent nodes not reachable along realizable paths. (For example, note that nodes $\langle n8, g \rangle$ and $\langle n9, g \rangle$ are reachable only along non-realizable paths from $\langle s_{main}, \mathbf{0} \rangle$.)

This information indicates the nodes' values in the meet-over-all-valid-paths solution to the dataflow-analysis problem. For instance, in the meet-over-all-valid-paths solution, $MVP_{e_p} = \{ g \}$. (That is, variable g is the only possibly-uninitialized variable just before execution reaches the exit node of procedure p.) In Figure 2, this information can be obtained by determining that there is a realizable path from $\langle s_{main}, \mathbf{0} \rangle$ to $\langle e_p, g \rangle$, but not from $\langle s_{main}, \mathbf{0} \rangle$ to $\langle e_p, a \rangle$. $\square$

4. An Efficient Algorithm for the Realizable-Path Reachability Problem

In this section, we present our algorithm for the realizable-path reachability problem. The algorithm is a dynamic-programming algorithm that tabulates certain kinds of same-level realizable paths. As discussed in Section 5 and the Appendix, the algorithm's running time is polynomial in various parameters of the problem, and it is asymptotically faster than the best previously known algorithm for the problem.

The algorithm, which we call the ***Tabulation Algorithm***, is presented in Figure 3. The algorithm uses the following functions:

- *returnSite*: maps a call node to its corresponding return-site node;
- *procOf*: maps a node to the name of its enclosing procedure;
- *calledProc*: maps a call node to the name of the called procedure;
- *callers*: maps a procedure name to the set of call nodes that represent calls to that procedure.

The Tabulation Algorithm uses a set named PathEdge to record the existence of ***path edges***, which represent a subset of the same-level realizable paths in graph $G^\#_{IP}$. In particular, the source of a path edge is always a node of the form $\langle s_p, d_1 \rangle$ such that a realizable path exists from node $\langle s_{main}, \mathbf{0} \rangle$ to $\langle s_p, d_1 \rangle$. In other words, a path edge from $\langle s_p, d_1 \rangle$ to $\langle n, d_2 \rangle$ represents the suffix of a realizable path from node $\langle s_{main}, \mathbf{0} \rangle$ to $\langle n, d_2 \rangle$.

The Tabulation Algorithm uses a set named SummaryEdge to record the existence ***summary edges***, which represent same-level realizable paths that run from nodes of the form $\langle n, d_1 \rangle$, where $n \in Call$, to $\langle returnSite(n), d_2 \rangle$. In terms of the dataflow problem being solved, summary edges represent (partial) information about how the dataflow value after a call depends on the dataflow value before the call.

The Tabulation Algorithm is a worklist algorithm that accumulates sets of path edges and summary edges. The initial set of path edges represents the 0-length same-level realizable path from $\langle s_{main}, \mathbf{0} \rangle$ to $\langle s_{main}, \mathbf{0} \rangle$ (see line [2]). On each iteration of the main loop in procedure ForwardTabulateSLRPs (lines [10]-[39]), the algorithm deduces the existence of additional path edges (and summary edges). The configurations that are used by the Tabulation Algorithm to deduce the existence of additional path edges are depicted in Figure 4.

Once it is known that there is a realizable path from $\langle s_{main}, \mathbf{0} \rangle$ to $\langle s_p, d \rangle$, path edge $\langle s_p, d \rangle \to \langle s_p, d \rangle$ is inserted into WorkList (lines [14]-[16]). In this case, path edge $\langle s_p, d \rangle \to \langle s_p, d \rangle$ represents the 0-length suffix of a realizable path from $\langle s_{main}, \mathbf{0} \rangle$ to $\langle s_p, d \rangle$. (The idea of inserting only relevant $\langle s_p, d \rangle \to \langle s_p, d \rangle$ edges into WorkList is similar to the idea of avoiding unnecessary function applications during abstract interpretation, known variously as "chaotic iteration with needed information only" [10] or the "minimal function-graph approach" [18].)

It is important to note the role of lines [26]-[28] of Figure 3, which are executed only when a new summary edge is discovered:

```
     declare PathEdge, WorkList, SummaryEdge: global edge set
     algorithm Tabulate(G^#_IP)
     begin
[1]    Let (N^#, E^#) = G^#_IP
[2]    PathEdge := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
[3]    WorkList := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
[4]    SummaryEdge := ∅
[5]    ForwardTabulateSLRPs()
[6]    for each n ∈ N^* do
[7]      X_n := { d_2 ∈ D | ∃ d_1 ∈ (D ∪ { 0 }) such that ⟨s_procOf(n), d_1⟩ → ⟨n, d_2⟩ ∈ PathEdge }
[8]    od
     end

     procedure Propagate(e)
     begin
[9]    if e ∉ PathEdge then  Insert e into PathEdge;  Insert e into WorkList  fi
     end

     procedure ForwardTabulateSLRPs()
     begin
[10]   while WorkList ≠ ∅ do
[11]     Select and remove an edge ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from WorkList
[12]     switch n
[13]       case n ∈ Call_p :
[14]         for each d_3 such that ⟨n, d_2⟩ → ⟨s_calledProc(n), d_3⟩ ∈ E^# do
[15]           Propagate(⟨s_calledProc(n), d_3⟩ → ⟨s_calledProc(n), d_3⟩)
[16]         od
[17]         for each d_3 such that ⟨n, d_2⟩ → ⟨returnSite(n), d_3⟩ ∈ (E^# ∪ SummaryEdge) do
[18]           Propagate(⟨s_p, d_1⟩ → ⟨returnSite(n), d_3⟩)
[19]         od
[20]       end case
[21]       case n = e_p :
[22]         for each c ∈ callers(p) do
[23]           for each d_4, d_5 such that ⟨c, d_4⟩ → ⟨s_p, d_1⟩ ∈ E^# and ⟨e_p, d_2⟩ → ⟨returnSite(c), d_5⟩ ∈ E^# do
[24]             if ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ ∉ SummaryEdge then
[25]               Insert ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ into SummaryEdge
[26]               for each d_3 such that ⟨s_procOf(c), d_3⟩ → ⟨c, d_4⟩ ∈ PathEdge do
[27]                 Propagate(⟨s_procOf(c), d_3⟩ → ⟨returnSite(c), d_5⟩)
[28]               od
[29]             fi
[30]           od
[31]         od
[32]       end case
[33]       case n ∈ (N_p − Call_p − { e_p }) :
[34]         for each ⟨m, d_3⟩ such that ⟨n, d_2⟩ → ⟨m, d_3⟩ ∈ E^# do
[35]           Propagate(⟨s_p, d_1⟩ → ⟨m, d_3⟩)
[36]         od
[37]       end case
[38]     end switch
[39]   od
     end
```

Figure 3. The Tabulation Algorithm determines the meet-over-all-valid-paths solution to *IP* by determining whether certain same-level realizable paths exist in $G^{\#}_{IP}$.

```
[26]  for each d_3 such that ⟨s_procOf(c), d_3⟩ → ⟨c, d_4⟩
          ∈ PathEdge do
[27]    Propagate(⟨s_procOf(c), d_3⟩ → ⟨returnSite(c), d_5⟩)
[28]  od
```

Unlike edges in $E^{\#}$, edges are inserted into SummaryEdge on-the-fly. The purpose of line [27] is to restart the processing that finds same-level realizable paths from $\langle s_{procOf(c)}, d_3 \rangle$ as if summary edge $\langle c, d_4 \rangle \rightarrow \langle returnSite(c), d_5 \rangle$ had been in place all along.

The final step of the Tabulation Algorithm (lines [6]-[8]) is to create values X_n, for each $n \in N^*$, by gathering up the set of nodes associated with n in $G^{\#}_{IP}$ that are targets of path edges discovered by procedure ForwardTabulateSLRPs:

```
[7]  X_n := { d_2 ∈ D | ∃ d_1 ∈ (D ∪ { 0 }) such that
          ⟨s_procOf(n), d_1⟩ → ⟨n, d_2⟩ ∈ PathEdge }
```

As mentioned above, the fact that edge $\langle s_{procOf(n)}, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ is in PathEdge implies that there is a realizable path from $\langle s_{main}, \mathbf{0} \rangle$ to $\langle n, d_2 \rangle$. Consequently, by Theorem 3.8, when the Tabulation Algorithm terminates, the value of X_n is the value for node n in the

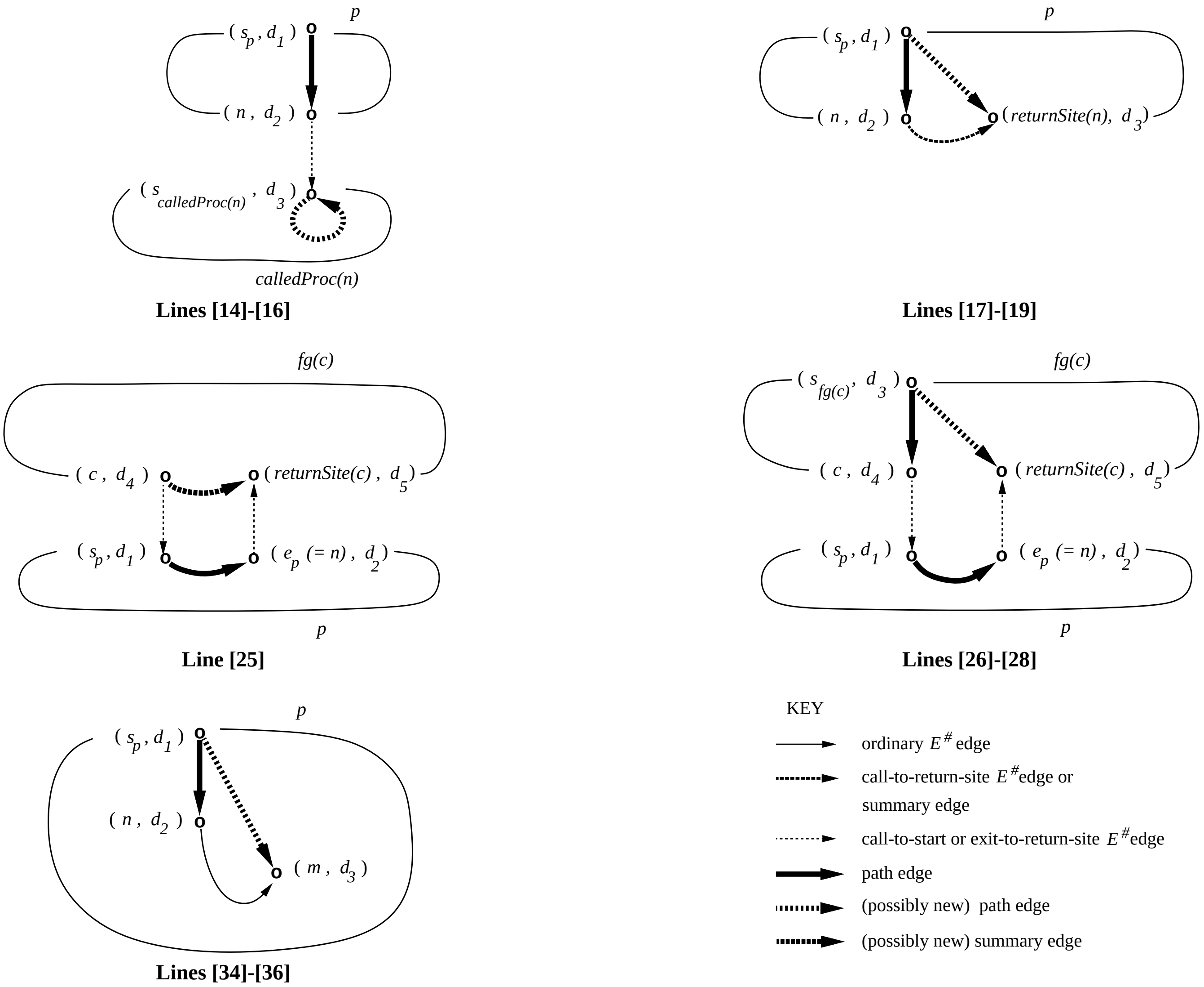


Figure 4. The above five diagrams show the situations handled in lines [14]-[16], [17]-[19], [25], [26]-[28], and [34]-[36] of the Tabulation Algorithm.

meet-over-all-valid-paths solution to *IP*.

Theorem 4.1. (Correctness of the Tabulation Algorithm.) *The Tabulation Algorithm always terminates, and upon termination, $X_n = MVP_n$, for all $n \in N^*$.*

5. The Cost of the Tabulation Algorithm

The running time of the Tabulation Algorithm varies depending on what class of dataflow-analysis problems it is applied to. We have already mentioned the locally separable problems; it is also useful to define the class of *h*-sparse problems:

Definition 5.1. A problem is ***h-sparse*** if all problem instances have the following property: For each function on an ordinary intraprocedural edge or a call-to-return-site edge, the total number of edges in the function's representation relation that emanate from the non-**0** nodes is at most hD. □

In general, when the nodes of the control-flow graph represent individual statements and predicates (rather than basic blocks), and there is no aliasing, we expect most distributive problems to be *h*-sparse (with $h \ll D$): each statement changes only a small portion of the execution state, and accesses only a small portion of the state as well. The dataflow functions, which are abstractions of the statements' semantics, should therefore be "close to" the identity function, and thus their representation relations should have roughly D edges. For many problems of practical interest $h \leq 2$ (see [27]).

Example. When the nodes of the control-flow graph represent individual statements and predicates, and there is no aliasing, every instance of the possibly-uninitialized-variables problem is 2-sparse. The only non-identity dataflow functions are those associated with assignment statements. The outdegree of every non-**0** node in the representation relation of such a function is at most two: a variable's initialization status can affect itself and at most one other variable, namely the variable assigned to. □

In analyzing the Tabulation Algorithm, we assume that all primitive set operations are unit-cost. This can be achieved, for instance, by the representation described in [27, pp. 20].

Table 5.2 summarizes how the Tabulation Algorithm behaves (in terms of worst-case asymptotic running time) for six different classes of problems:

Class of dataflow functions	Characterization of the functions' properties	Asymptotic running time: Intra-procedural problems	Asymptotic running time: Inter-procedural problems
Distributive	Up to $O(D^2)$ edges/rep.-relation	$O(ED^2)$	$O(ED^3)$
h-sparse	At most $O(hD)$ edges/rep.-relation	$O(hED)$	$O(Call\ D^3 + hED^2)$
(Locally) separable	Component-wise dependences	$O(ED)$	$O(ED)$

Table 5.2. Asymptotic running time of the Tabulation Algorithm for six different classes of dataflow-analysis problems.

The details of the analysis of the running time of the Tabulation Algorithm on distributive problems are given in the Appendix. The bounds for the other five classes of problems follow from simplifications of the argument given there.

The storage requirements for the Tabulation Algorithm consist of the storage for graph $G^{\#}_{IP}$ and the three sets WorkList, PathEdge, and SummaryEdge, which are bounded by $O(ED^2)$, $O(ND^2)$, $O(ND^2)$ and $O(Call\ D^2)$.

6. Preliminary Experimental Results

We have carried out a preliminary study to determine the feasibility of the Tabulation Algorithm. In the study, we compared the algorithm's accuracy and time requirements with those of the safe, but naive, reachability algorithm that considers *all* paths in the exploded supergraph, rather than just the realizable paths. The two algorithms were implemented in C and used with a front end that analyzes a C program and generates the corresponding exploded supergraph for the possibly-uninitialized-variables problem. (The current implementation of the front end does not account for aliases due to pointers.)

The study used four example C programs: *struct-beauty*, the "beautification" phase of the Unix *struct* program [3]; *twig*, a code-generator generator [2]; *ratfor*, a preprocessor that converts a structured Fortran dialect to standard Fortran [19]; and *C-parser*, a *lex*/*yacc*-generated parser for C. Tests were carried out on a Sun SPARCstation 10 Model 30 with 32 MB of RAM.

The following table gives information about the source code (lines of C, *lex*, and *yacc*) and the parameters that characterize the size of the control-flow graphs and the exploded supergraph.

Example	Lines of source code	CFG statistics: *P*	*Call*	*N*	*E*	$G^{\#}$ statistics: *D*	$N^{\#}$	$E^{\#}$
struct-beauty	897	36	214	2188	2860	90	183.9k	220.6k
C-parser	1224	48	78	1637	1992	70	104.4k	112.4k
ratfor	1345	52	266	2239	2991	87	179.5k	217.7k
twig	2388	81	221	3692	4439	142	492.2k	561.1k

In practice, most of the $E^{\#}$ edges are of the form $\langle m, d\rangle \rightarrow \langle n, d\rangle$, and our implementation takes advantage of this to represent these edges in a compact way.

The following table compares the cost and accuracy of the Tabulation Algorithm and the naive algorithm. The running times are "user cpu-time + system cpu-time"; in each case, the time reported is the average of ten executions.

Example	Tabulation Algorithm (realizable paths): Time (sec.)	Reported uses of possibly uninitialized variables	Naive Algorithm (any path): Time (sec.)	Reported uses of possibly uninitialized variables
struct-beauty	4.83+0.75	543	1.58+0.04	583
C-parser	0.70+0.19	11	0.54+0.02	127
ratfor	3.15+0.58	894	1.46+0.04	998
twig	5.45+1.20	767	5.04+0.11	775

The number of uses of possibly-uninitialized variables reported by the Tabulation Algorithm ranges from 9% to 99% of those reported by the naive algorithm. Because the possibly-uninitialized-variables problem is 2-sparse, the asymptotic costs of the Tabulation Algorithm and the naive algorithm are $O(Call\ D^3 + ED^2)$ and $O(ED)$, respectively. In these examples, *D* ranges from 70 to 142; however, the penalty for obtaining the more precise solutions ranges from 1.3 to 3.4. Therefore, this preliminary experiment suggests that the extra precision of meet-over-all-valid-paths solutions to interprocedural dataflow-analysis problems can be obtained by the Tabulation Algorithm with acceptable cost.

7. Related Work

Previous Interprocedural Dataflow-Analysis Frameworks

The IFDS framework is based on earlier interprocedural dataflow-analysis frameworks defined by Sharir and Pnueli [31] and Knoop and Steffen [21]. It is basically the Sharir-Pnueli framework with three modifications:

(i) The dataflow domain is restricted to be a subset domain 2^D, where *D* is a finite set;
(ii) The dataflow functions are restricted to be distributive functions;
(iii) The edge from a call node to the corresponding return-site node can have an associated dataflow function.

Conditions (i) and (ii) are restrictions that make the IFDS framework less general than the full Sharir-Pnueli framework. Condition (iii), however, generalizes the Sharir-

Pnueli framework and permits it to cover programming languages in which recursive procedures have local variables and parameters (which the Sharir-Pnueli framework does not). (A different generalization to handle recursive procedures with local variables and parameters was proposed by Knoop and Steffen [21].)

The IFDS problems can be solved by a number of previous algorithms, including the "elimination", "iterative", and "call-strings" algorithms given by Sharir and Pnueli [31] and the algorithm of Cousot and Cousot [10]. However, for general IFDS problems both the iterative and call-strings algorithms can take exponential time in the worst case. Knoop and Steffen give an algorithm similar to Sharir and Pnueli's "elimination" algorithm [21]. The efficiencies of the Sharir-Pnueli and Knoop-Steffen elimination algorithms depend, among other things, on the way functions are represented. No representations are discussed in [31] and [21]. However, even if representation relations (as defined in Section 3.1) are used, because the Sharir-Pnueli and Knoop-Steffen algorithms manipulate functions as a whole, rather than pointwise, for distributive and h-sparse problems, they are not as efficient as the Tabulation Algorithm.

Holley and Rosen investigated "qualified" dataflow analysis problems, where "qualifications" are a device to specify that only certain paths in the flow graph are to be considered [15]. They employ an "expansion" phase that has some similarities to our creation of the exploded supergraph. However, Holley and Rosen do not take advantage of distributivity to do the expansion pointwise, and thus for the IFDS problems they would create 2^D points per flow-graph node, as opposed to the D points used in our approach. Furthermore, for interprocedural problems the Holley-Rosen approach is equivalent to the (impractical) Sharir-Pnueli call-strings approach.

Reps investigated the use of deductive databases (*i.e.*, logic programs with a bottom-up evaluation engine) to implement locally separable interprocedural dataflow-analysis problems [29]. This approach can be viewed as a pointwise tabulation method. Although the present paper does not make use of logic-programming terminology, the Tabulation Algorithm has a straightforward implementation as a logic program. Thus, another contribution of the present paper is that it shows how to extend the logic-programming approach from the class of locally separable problems to the class of IFDS problems.

Dataflow Analysis via Graph Reachability and Pointwise Computation of Fixed Points

Our work shows that a large subclass of the problems in the Sharir-Pnueli and Knoop-Steffen frameworks can be posed as graph-reachability problems. Other work on solving dataflow-analysis problems by reducing them to reachability problems has been done by Kou [23] and Cooper and Kennedy [7,8]. In each case a dataflow-analysis problem is solved by first building a graph—derived from the program's flow graph and the dataflow functions to be solved—and then performing a reachability analysis on the graph by propagating simple marks. (This contrasts with standard iterative techniques, which propagate sets of values over the flow graph.)

Kou's paper addresses only intraprocedural problems. Although he only discusses the live-variable problem, his ideas immediately carry over to all the separable intraprocedural problems. Cooper and Kennedy show how certain flow-insensitive interprocedural dataflow-analysis problems can be converted to reachability problems. Because they deal only with flow-insensitive problems, the solution method involves ordinary reachability rather than the more difficult question of reachability along realizable paths.

Zadeck developed intraprocedural dataflow analysis algorithms based on the idea of partitioning a problem into many independent problems (*e.g.*, on a "per-bit" basis in the case of separable problems) [32]. Although our technique of "exploding" a problem into the exploded supergraph transforms locally separable problems into a number of independent "per-fact" subproblems, the technique does not yield *independent* subproblems for h-sparse and general distributive IFDS problems. For example, in the 2-sparse possibly-uninitialized variables problem, a given variable may be transitively affected by *any* of the other variables. Nevertheless, these problems can be solved efficiently by the Tabulation Algorithm.

Graph reachability can also be thought of as an implementation of the pointwise computation of fixed points, which has been studied by Cai and Paige [4] and Nielson and Nielson [26,25]. Theorem 3.3, the basis on which we convert dataflow-analysis problems to reachability problems, is similar to Lemma 14 of Cai and Paige; however, the relation that Cai and Paige define for representing distributive functions does not have the subsumption property. Although it does not change the asymptotic complexity of the Tabulation Algorithm, using relations that have the subsumption property decreases the number of edges in the exploded supergraph and consequently reduces the running time of the Tabulation Algorithm.

Cai and Paige show that pointwise computation of fixed points can be used to compile programs written in a very-high-level language (SQ+) into efficient executable code. This suggests that it might be possible to express the problem of finding meet-over-all-valid-paths solutions to IFDS problems as an SQ+ fixed-point program and then automatically compile it into an implementation that achieves the bounds established in this paper (*i.e.*, into the Tabulation Algorithm).

Nielson and Nielson investigated bounds on the cost of a general fixed-point-finding algorithm by computing the cost as "(# of iterations) $\times$ (cost per iteration)". Their main contribution was to give formulas for bounding the number of iterations based on properties of both the functional and the domain in which the fixed-point is computed. Their formula for "strict and additive" functions can be adapted to our context of (non-strict) distributive functions, and used to show that the number of iterations of the Tabulation Algorithm is at most ND^2. The cost of a single iteration can be $O(Call\,D^2 + kD^2)$, where k is the maximum outdegree of a node in the control-flow graph. Thus, this approach gives a bound for the total cost of the Tabulation Algorithm of $O((ND^2) \times (Call\,D^2 + kD^2)) = O(Call\,ND^4 + kND^4)$, which compares unfavorably with our bound of $O(ED^3)$.

In contrast, the bound that we have presented for the cost of the Tabulation Algorithm is obtained by breaking the cost of the algorithm into three contributing aspects and bounding the *total* cost of the operations performed for each aspect (see the Appendix).

Another example of pointwise tabulation is Landi and Ryder's algorithm for interprocedural alias analysis for single-level pointers [24]. The algorithm they give is similar to the Tabulation Algorithm. A limitation of the IFDS framework is that information at a return-site node can only be expressed as the meet of the information at the corresponding call node and the appropriate exit node. Because in the single-level-pointer problem the combining function for return-site nodes is not meet, the problem does not fit into the IFDS framework.

Flow-Sensitive Side-Effect Analysis

Callahan investigated two flow-sensitive side-effect problems: must-modify and may-use [6]. The must-modify problem is to identify, for each procedure p, which variables must be modified during a call on p; the may-use problem is to identify, for each procedure p, which variables may be used before being modified during a call on p. Callahan's method involves building a *program summary graph*, which consists of a collection of graphs that represent the intraprocedural reaching-definitions information between start, exit, call, and return-site nodes—together with interprocedural linkage information.[1]

Although the must-modify and may-use problems are not IFDS problems as defined in Definition 2.4, they can be viewed as problems closely related to the IFDS problems. The basic difference is that IFDS problems summarize what must be true at a program point in *all calling contexts*, while the must-modify and may-use problems summarize the effects of a procedure *isolated from its calling contexts*. That is, Callahan's problems involve valid paths from the individual procedures' start nodes rather than just the start node of the main procedure. The must-modify problem is actually a "*same-level*-valid-path" problem rather than a "valid-path" problem; the must-modify value for each procedure involves only the same-level valid paths from the procedure's start node to its exit node. Consequently, Callahan's problems can be thought of as examples of problems in two more general *classes* of problems: a class of distributive valid-path problems, and a class of distributive same-level valid-path problems.

The method utilized in the present paper is to convert distributive valid-path dataflow-analysis problems into realizable-path reachability problems in an exploded supergraph. By transformations analogous to the one given in Section 3,

(i) the distributive valid-path problems can be posed as realizable-path problems;
(ii) the distributive same-level valid-path problems can be posed as same-level realizable-path problems.

In particular, the may-use problem is a locally separable problem in class (i); the must-modify problem is a locally separable problem in class (ii).

The payoff from adopting this generalized viewpoint is that, with only slight modifications, the Tabulation Algorithm can be used to solve *all* problems in the above two classes (*i.e.*, distributive and h-sparse problems, as well as the locally separable ones). The modified algorithms have the same asymptotic running time as the Tabulation Algorithm. In particular, for the locally separable problems—such as must-modify and may-use—the running time is bounded by $O(ED)$. This is an asymptotic improvement over the algorithms given by Callahan: the worst-case cost for building the program summary graph is $O(D \sum_p Call_p E_p)$; given the program summary graph, the worst-case cost for computing must-modify or may-use is $O(D \sum_p Call_p^2)$.

Demand Algorithms for Interprocedural Dataflow Analysis

The goal of demand dataflow analysis is to determine whether a given dataflow fact holds at a given point (while minimizing the amount of auxiliary dataflow information computed for other program points). One of the benefits of the IFDS framework is that it permits a simple implementation of a demand algorithm for interprocedural dataflow analysis [27,17].

Other work on demand interprocedural dataflow analysis includes [29] and [11].

The IDE Framework

Recently, we generalized the IFDS framework to a larger class of problems, called the IDE framework. In the IDE framework, the dataflow facts are maps ("environments") from some finite set of symbols to some (possibly infinite) set of values, and the dataflow functions are distributive environment transformers [30]. ("IDE" stands for ***I***nterprocedural ***D***istributive ***E***nvironment problems.) The IDE problems are a proper superset of the IFDS problems in that there are certain IDE problems (including variants of interprocedural constant propagation) that cannot be encoded as IFDS problems.

Although the transformation we apply to IDE problems is similar to the one used for IFDS problem, the transformed problem that results is a realizable-path *summary* problem, not a realizable-path *reachability* problem. That is, in the transformed graph we are no longer concerned with a pure reachability problem, but with values obtained by applying functions along (realizable) paths. (The relationship between transformed IFDS problems and transformed IDE problems is similar to the relationship between ordinary graph-reachability problems and generalized problems that compute summaries over paths, such as shortest-path problems, closed-semiring path problems, *etc.* [1].) The algorithm for solving IDE problems is a dynamic-programming algorithm similar to the Tabulation Algorithm.

Appendix: The Running Time of the Tabulation Algorithm

In this section, we present a derivation of the bound given in Table 5.2 for the cost of the Tabulation Algorithm on distributive problems.

Instead of calculating the worst-case cost-per-iteration of the loop on lines [10]-[39] of Figure 3 and multiplying by the number of iterations, we break the cost of the algorithm down into three contributing aspects and bound the *total* cost of the operations performed for each aspect. In particular, the cost of the Tabulation Algorithm can be broken down into

[1]Although the equations that Callahan gives contain both $\wedge$ and $\vee$ operators, this is not because his problems are some kind of "heterogeneous meet/join problems". For example, when Callahan's flow-sensitive Kill problem is reformulated in the Sharir-Pnueli framework, $\wedge$ corresponds to meet, but $\vee$ corresponds to *composition* of edge functions.

(i) the cost of worklist manipulations,
(ii) the cost of installing summary edges at call sites (lines [21]-[32] of Figure 3), and
(iii) the cost of “closure” steps (lines [13]-[20] and [33]-[37] of Figure 3).

Because a path edge can be inserted into WorkList at most once, the cost of each worklist-manipulation operation can be charged to either a summary-edge-installation step or a closure step; thus, we do not need to provide a separate accounting of worklist-manipulation costs.

The Tabulation Algorithm can be understood as $k+1$ simultaneous *semi-dynamic multi-source reachability problems*—one per procedure of the program. For each procedure p, the sources—which we shall call ***anchor sites***—are the $D+1$ nodes in $N^{\#}$ of the form $\langle s_p, d \rangle$. The edges of the multi-source reachability problem associated with p are

$$\{ \langle m, d_1 \rangle \rightarrow \langle n, d_2 \rangle \in E^{\#} \mid m, n \in N_p \text{ and } m \rightarrow n \text{ is an intraprocedural edge or a call-to-return-site edge} \}$$
$$\cup \{ \langle m, d_1 \rangle \rightarrow \langle n, d_2 \rangle \in \text{SummaryEdge} \mid m \in Call_p \}.$$

In other words, the graph associated with procedure p is the “exploded flow graph” of procedure p, augmented with summary edges at the call sites of p. The reachability problems are semi-dynamic (insertions only) because in the course of the algorithm, new summary edges are added, but no summary edges (or any other edges) are ever removed.

We first turn to the question of computing a bound on the cost of installing summary edges at call sites (lines [21]-[32] of Figure 3). To express this bound, it is useful to introduce a quantity B that represents the “bandwidth” for the transmission of dataflow information between procedures: In particular, B is the maximum value for all call-to-start edges and exit-to-return-site edges of (i) the maximum outdegree of a non-$\mathbf{0}$ node in a call-to-start edge’s representation relation; (ii) the maximum indegree of a non-$\mathbf{0}$ node in an exit-to-return-site edge’s representation relation. (In the worst case, B is D, but it is typically a small constant, and for many problems it is 1.)

For each summary edge $\langle c, d_4 \rangle \rightarrow \langle returnSite(c), d_5 \rangle$, the conditional statement on lines [24]-[29] will be executed some number of times (on different iterations of the loop on lines [10]-[39]). In particular, line [24] will be executed every time the Tabulation Algorithm finds a three-edge path of the form

$$[\langle c, d_4 \rangle \rightarrow \langle s_p, d_1 \rangle,\ \langle s_p, d_1 \rangle \rightarrow \langle e_p, d_2 \rangle,\ \langle e_p, d_2 \rangle \rightarrow \langle returnSite(c), d_5 \rangle] \qquad (\dagger)$$

as shown in the diagram marked “Line [25]” of Figure 4.

When we consider the set of all summary edges at a given call site c: $\{ \langle c, d_4 \rangle \rightarrow \langle returnSite(c), d_5 \rangle \}$, the executions of line [24] can be placed in three categories:

$d_4 \neq \mathbf{0}$ and $d_5 \neq \mathbf{0}$
: There are at most D^2 choices for a (d_4, d_5) pair, and for each such pair at most B^2 possible three-edge paths of the form (†).

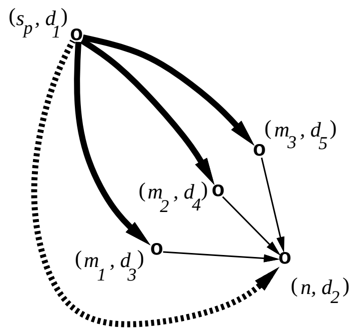
$d_4 = \mathbf{0}$ and $d_5 \neq \mathbf{0}$
: There are at most D choices for d_5 and for each such choice at most BD possible three-edge paths of the form (†).

$d_4 = \mathbf{0}$ and $d_5 = \mathbf{0}$
: There is only one possible three-edge path of the form (†).

Thus, the total cost of all executions of line [24] is bounded by $O(Call\, B^2 D^2)$.

Because of the test on line [24], the code on lines [25]-[28] will be executed exactly *once* for each possible summary edge. In particular, for each summary edge the cost of the loop on lines [26]-[28] is bounded by $O(D)$. Since the total number of summary edges is bounded by $Call\, D^2$, the total cost of lines [25]-[28] is $O(Call\, D^3)$. Thus, the total cost of installing summary edges during the Tabulation Algorithm is bounded by $O(Call\, B^2 D^2 + Call\, D^3)$.

To bound the total cost of the closure steps, the essential observation is that there are only a certain number of “attempts” the Tabulation Algorithm makes to “acquire” a path edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$. The first attempt is successful—and $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ is inserted into PathEdge; all remaining attempts are redundant (but seem unavoidable). In particular, in the case of a node $n \notin Ret$, the only way the Tabulation Algorithm can obtain a path edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ is when there are one or more two-edge paths of the form $[\langle s_p, d_1 \rangle \rightarrow \langle m, d \rangle,\ \langle m, d \rangle \rightarrow \langle n, d_2 \rangle]$, where $\langle s_p, d_1 \rangle \rightarrow \langle m, d \rangle$ is in PathEdge and $\langle m, d \rangle \rightarrow \langle n, d_2 \rangle$ is in $E^{\#}$, as depicted below:



Consequently, for a given anchor site $\langle s_p, d_1 \rangle$, the cost of the closure steps involved in acquiring path edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ can be bounded by $indegree(\langle n, d_2 \rangle)$. For distributive problems, the representation relation of the function on an ordinary intraprocedural edge or a call-to-return-site edge can contain up to $O(D^2)$ edges. Thus, for each anchor site, the total cost of acquiring all its outgoing path edges can be bounded by

$$O\Big(\sum_{\langle n, d \rangle \in N_p^{\#} \text{and } n \notin Ret} indegree(\langle n, d \rangle) \Big) = O(E_p D^2).$$

The accounting for the case of a node $n \in Ret$ is similar. The only way the Tabulation Algorithm can obtain a path edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ is when there is an edge in PathEdge of the form $\langle s_p, d_1 \rangle \rightarrow \langle m, d \rangle$ and either there is an edge $\langle m, d \rangle \rightarrow \langle n, d_2 \rangle$ in $E^{\#}$ or an edge $\langle m, d \rangle \rightarrow \langle n, d_2 \rangle$ in SummaryEdge. In our cost accounting, we will pessimistically assume that each node $\langle n, d_2 \rangle$, where $n \in Ret$, has the maximum possible number of incoming summary edges, namely D. Because there are at most $Call_p D$ nodes of $N_p^{\#}$ of the form $\langle n, d_2 \rangle$, where $n \in Ret$, for each anchor site $\langle s_p, d_1 \rangle$ the total cost of acquiring path edges of the form $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ is

$$O\Big(\sum_{\substack{\langle n, d_2 \rangle \in N_p^{\#} \\ \text{and } n \in Ret}} indegree(\langle n, d_2 \rangle) + summary\text{-}indegree(\langle n, d_2 \rangle) \Big)$$

which equals $O(Call_p D^2)$.

Therefore we can bound the total cost of the closure steps performed by the Tabulation Algorithm as follows:

Cost of closure steps

$$= \sum_p (\#\text{ anchor sites}) \times O(Call_p D^2 + E_p D^2)$$
$$= O(D^3 \sum_p (Call_p + E_p))$$
$$= O(D^3 (Call + E))$$
$$= O(ED^3).$$

Thus, the total running time of the Tabulation Algorithm is bounded by $O(Call\,B^2 D^2 + ED^3)$.

It is possible to improve this bound to $O(Call\,BD^2 + ED^3)$ by treating procedure linkages as if they were (B-sparse) procedures in their own right and introducing new linkages to the linkage procedures with "bandwidth" 1. Because $Call \le E$ and $B \le D$, this simplifies to $O(ED^3)$, the bound reported in Table 5.2.

References

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA (1974).
2. Aho, A.V., Ganapathi, M., and Tjiang, S.W.K., "Code generation using tree matching and dynamic programming," *ACM Trans. Program. Lang. Syst.* **11**(4) pp. 491-516 (October 1989).
3. Baker, B., "An algorithm for structuring flowgraphs," *J. ACM* **24**(1) pp. 98-120 (January 1977).
4. Cai, J. and Paige, R., "Program derivation by fixed point computation," *Science of Computer Programming* **11** pp. 197-261 (1988/89).
5. Callahan, D., Cooper, K.D., Kennedy, K., and Torczon, L., "Interprocedural constant propagation," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction,* (Palo Alto, CA, June 25-27, 1986), *ACM SIGPLAN Notices* **21**(7) pp. 152-161 (July 1986).
6. Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 47-56 (July 1988).
7. Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 57-66 (July 1988).
8. Cooper, K.D. and Kennedy, K., "Fast interprocedural alias analysis," pp. 49-59 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
9. Cousot, P. and Cousot, R., "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," pp. 238-252 in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages,* (Los Angeles, CA, January 17-19, 1977), ACM, New York, NY (1977).
10. Cousot, P. and Cousot, R., "Static determination of dynamic properties of recursive procedures," pp. 237-277 in *Formal Descriptions of Programming Concepts,* (IFIP WG 2.2, St. Andrews, Canada, August 1977), ed. E.J. Neuhold,North-Holland, New York, NY (1978).
11. Duesterwald, E., Gupta, R., and Soffa, M.L., "Demand-driven computation of interprocedural data flow," in *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages,* (San Francisco, CA, Jan. 23-25, 1995), ACM, New York, NY (1995). (To appear.)
12. Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler,* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).
13. Giegerich, R., Moncke, U., and Wilhelm, R., "Invariance of approximative semantics with respect to program transformation," pp. 1-10 in *Informatik-Fachberichte 50,* Springer-Verlag, New York, NY (1981).
14. Grove, D. and Torczon, L., "Interprocedural constant propagation: A study of jump function implementation," pp. 90-99 in *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation,* (Albuquerque, NM, June 23-25, 1993), ACM, New York, NY (1993).
15. Holley, L.H. and Rosen, B.K., "Qualified data flow problems," *IEEE Transactions on Software Engineering* **SE-7**(1) pp. 60-78 (January 1981).
16. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).
17. Horwitz, S., Reps, T., and Sagiv, M., "Demand interprocedural dataflow analysis," Unpublished Report, Computer Sciences Department, University of Wisconsin, Madison, WI (). (In preparation.)
18. Jones, N.D. and Mycroft, A., "Data flow analysis of applicative programs using minimal function graphs," pp. 296-306 in *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages,* (St. Petersburg, FL, Jan. 13-15, 1986), ACM, New York, NY (1986).
19. Kernighan, B.W., "Ratfor – A preprocessor for a rational Fortran," *Software – Practice & Experience* **5**(4) pp. 395-406 (1975).
20. Kildall, G., "A unified approach to global program optimization," pp. 194-206 in *Conference Record of the First ACM Symposium on Principles of Programming Languages,* ACM, New York, NY (1973).
21. Knoop, J. and Steffen, B., "The interprocedural coincidence theorem," pp. 125-140 in *Proceedings of the Fourth International Conference on Compiler Construction,* (Paderborn, FRG, October 5-7, 1992), *Lecture Notes in Computer Science,* Vol. 641, ed. U. Kastens and P. Pfahler,Springer-Verlag, New York, NY (1992).
22. Knoop, J. and Steffen, B., "Efficient and optimal bit-vector data flow analyses: A uniform interprocedural framework," Bericht Nr. 9309, Institut fuer Informatik und Praktische Mathematik, Christian-Albrechts-Universitaet zu Kiel, Kiel, Germany (April 1993).
23. Kou, L.T., "On live-dead analysis for global data flow problems," *Journal of the ACM* **24**(3) pp. 473-483 (July 1977).
24. Landi, W. and Ryder, B.G., "Pointer-induced aliasing: A problem classification," pp. 93-103 in *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages,* (Orlando, FL, January 1991), ACM, New York, NY (1991).
25. Nielson, F. and Nielson, H.R., "Finiteness conditions for fixed point iteration," in *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming,* (San Francisco, CA, June 22-24 1992), ACM, New York, NY (1992).
26. Nielson, H.R. and Nielson, F., "Bounded fixed point iteration," pp. 71-82 in *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages,* (Albuquerque, NM, January 1992), ACM, New York, NY (1992).
27. Reps, T., Sagiv, M., and Horwitz, S., "Interprocedural dataflow analysis via graph reachability," TR 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark (April 1994). (Available through World Wide Web at ftp://ftp.diku.dk/diku/semantics/papers/D-215.ps.Z.)
28. Reps, T., Horwitz, S., Sagiv, M., and Rosay, G., "Speeding up slicing," *SIGSOFT 94: Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (New Orleans, LA, December 7-9, 1994), *ACM SIGSOFT Software Engineering Notes* **19**(December 1994). (To appear.)
29. Reps, T., "Solving demand versions of interprocedural analysis problems," pp. 389-403 in *Proceedings of the Fifth International Conference on Compiler Construction,* (Edinburgh, Scotland, April 7-9, 1994), *Lecture Notes in Computer Science,* Vol. 786, ed. P. Fritzson,Springer-Verlag, New York, NY (1994).
30. Sagiv, M., Reps, T., and Horwitz, S., "Precise interprocedural dataflow analysis with applications to constant propagation," Unpublished Report, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI (Oct. 1994). (Submitted for conference publication.)
31. Sharir, M. and Pnueli, A., "Two approaches to interprocedural data flow analysis," pp. 189-233 in *Program Flow Analysis: Theory and Applications,* ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).
32. Zadeck, F.K., "Incremental data flow analysis in a structured program editor," *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction,* (Montreal, Can., June 20-22, 1984), *ACM SIGPLAN Notices* **19**(6) pp. 132-143 (June 1984).